Instruction-Driven Development (IDD) by Michaël Bettan

01. The Problem with Ad-Hoc Prompting

Most developers using AI coding assistants start in an **ad-hoc, reactive mode**: they open a chat, type a prompt like *"build me a login page,"* accept the output, and move to the next thing. This works for simple tasks but breaks down for anything complex, team-based, or production-grade.

The result? **Inconsistent outputs**, lost context between sessions, no reproducibility, and code that works in isolation but fails when integrated. IDD solves this by treating AI interaction as a *structured engineering process*, not a casual conversation.

Inconsistency

The same prompt produces different results across sessions, models, or team members. No deterministic outputs.

Context Loss

AI forgets previous decisions. Each session starts from zero. Architectural coherence degrades over time.

No Collaboration

Prompts live in one developer's head. No shared understanding, no review process, no handoff protocol.

> THE AD-HOC SPIRAL
> Without structure, each AI interaction accumulates subtle inconsistencies. By the time you notice, your codebase is a patchwork of individually-correct but collectively-incoherent components. This is the **"death by a thousand prompts"** anti-pattern.

02. What is IDD / SDD?

Instruction-Driven Development (IDD) — also called **Specification-Driven Development (SDD)** — is a methodology where you write detailed, structured instructions *before* engaging the AI. These instructions serve as the contract between human intent and AI execution.

Think of it as **TDD (Test-Driven Development) but for prompts**: instead of writing tests first then code, you write specifications first then let AI generate the implementation. The spec is the single source of truth — versioned, reviewed, and shared across the team.

Core Philosophy of IDD

- **Spec First, Code Second:** Write the instruction document before touching the AI — clarify what you want before asking for it
- **Instructions as Artifacts:** Specs are version-controlled, peer-reviewed, and reusable — they're as important as the code itself
- **Deterministic Intent:** The same instruction should produce functionally equivalent results regardless of when, where, or by whom it's executed
- **Separation of Concerns:** Human handles *what* and *why*; AI handles *how*
- **Iterative Refinement:** Specs evolve through feedback loops — each iteration makes instructions more precise and AI output more reliable

> IDD VS SDD — SAME THING, DIFFERENT NAMES
> The community uses both terms interchangeably. **IDD (Instruction-Driven Development)** emphasizes the instructional nature of prompts. **SDD (Specification-Driven Development)** emphasizes the formal specification. In practice, they describe the identical workflow.

03. The IDD Workflow (Step-by-Step)

IDD follows a disciplined, six-phase workflow (`Define → Review → Execute → Validate → Refine → Archive`). Each phase has clear inputs, outputs, and quality gates.

Detailed Phase Breakdown

1. **Define the Instruction:** Write a structured document containing: project context, functional requirements, non-functional requirements (performance, security), technical constraints (language, framework, patterns), file structure expectations, test requirements, and acceptance criteria.
2. **Review & Validate the Spec:** Have a teammate or second AI review the instruction for ambiguity, missing edge cases, contradictions, and completeness (e.g., via Antigravity `/plan` Artifact review or `/grill-me` design interviews). Fix before executing.
3. **Execute with AI:** Feed the instruction to your AI assistant as primary context alongside relevant existing files. Ask the AI to confirm its plan before generating code.
4. **Validate Output Against Spec:** Check every acceptance criterion. Run generated unit/integration tests and eval sets (`agents-cli eval run`). Verify file structure matches spec and scan for security issues.
5. **Refine the Instruction:** Based on what the AI got right and wrong, update the spec to remove ambiguity and add explicit constraints where the model drifted.
6. **Archive & Reuse:** Commit the instruction alongside the code in the same branch/PR. Promote repeatable workflows into modular `SKILL.md` packages or template libraries.

> THE GOLDEN RULE OF IDD
> If your instruction is ambiguous, the AI's output will be unpredictable. The quality of your specification directly determines the quality of the generated code. **Invest time in the spec, save time in debugging.**

04. Instruction Documents (Spec Anatomy)

The instruction document is the heart of IDD — a **structured specification** with six canonical sections:

```markdown
# Instruction: [Feature Name]
```

Version: 1.0 | Date: 2026-04-15 | Author: [Name]

1. Context

Project: [Project name and brief description] Current State: [What exists, what's the starting point] Goal: [What this instruction should produce]

2. Functional Requirements

- FR-01: User can register with email and password
- FR-02: System sends verification email on registration
- FR-03: User can login and receive a JWT token

3. Non-Functional Requirements

- NFR-01: Response time < 200ms for auth endpoints
- NFR-02: Passwords hashed with bcrypt (cost factor 12)
- NFR-03: Rate limiting: 5 login attempts per minute per IP

4. Technical Constraints

- Language: Python 3.12
- Framework: FastAPI
- Database: PostgreSQL via SQLAlchemy
- Auth: JWT with PyJWT
- Testing: pytest with >80% coverage

5. File Structure

src/ auth/ router.py # API endpoints service.py # Business logic models.py # SQLAlchemy models schemas.py # Pydantic schemas tests/ test_auth.py # Unit + integration tests

6. Acceptance Criteria

- □ All FR items implemented and tested
- □ All NFR constraints verified
- □ No security vulnerabilities (no hardcoded secrets)
- □ Code passes ruff linting with zero warnings
- □ README updated with setup instructions ```
- **Context Block:** Gives the AI project awareness — what exists, what's the goal, and what constraints are inherited from previous decisions.
- **Functional Requirements:** Numbered, independently testable statements of what the system must do.
- **Non-Functional Requirements:** Latency, security, scalability, and quality guardrails that prevent shortcuts.
- **Technical Constraints:** Locked-in language versions, frameworks, databases, and testing libraries.
- **File Structure:** Explicit directory and file layout preventing ad-hoc file sprawl.
- **Acceptance Criteria:** Verifiable pass/fail conditions for "done."

05. IDD vs Vibe Coding vs Traditional

Dimension	Traditional Dev	Vibe Coding	IDD / SDD
Planning	Requirements doc / tickets	Mental model + ad-hoc prompts	Structured instruction document + Plan Artifact
Implementation	Human writes all code	AI generates, human reviews	AI generates from spec, human + CI gates validate
Reproducibility	High (deterministic)	Low (prompt-dependent)	High (spec + evalset form the contract)
Team Scalability	High (established processes)	Low (tribal knowledge)	High (shared specs, rules & `SKILL.md` registry)
Speed	Slow (manual coding)	Very fast (but fragile)	Fast (after upfront spec investment)
Quality Control	Code review + CI	Varies (often minimal)	Spec review + deterministic hooks + eval CI
Best For	Critical systems, large teams	Prototypes, solo exploration	Production AI-assisted software engineering

IDD IS VIBE CODING'S ADULT FORM
IDD doesn't replace vibe coding — it **matures** it. Teams explore ideas rapidly with vibe coding, then formalize architecture, constraints, and acceptance tests into IDD specs for production delivery (`Vibe Coding = Prototyping → IDD = Engineering`).

06. Real-World IDD Patterns

Pattern 1: Layered Specs (`L0 → L3`)

- **L0 — Project Spec:** Overall architecture, tech stack, security invariants (`AGENTS.md` / `CLAUDE.md`)
- **L1 — Module Specs:** One per domain module (`auth` , `billing` , `notifications`)
- **L2 — Feature Specs:** Individual user-facing features within a module
- **L3 — Task Specs:** Atomic coding tasks (add endpoint, migrate schema, fix bug)
- Each lower layer inherits constraints from the layer above without duplicating boilerplate.

Pattern 2: Spec-Test-Code Loop

1. Write the instruction spec with explicit acceptance criteria and eval cases.
2. Generate tests (and `*.evalset.json` cases for agentic features) from the spec *first*.
3. Generate the implementation until all tests and evaluation rubrics pass.
4. Refine either the spec or implementation whenever edge cases surface.

Pattern 3: Spec as PR Template

- Every pull request links or embeds the IDD spec that produced the diff.
- Reviewers verify implementation fidelity against the spec, preserving architectural *why* in version history.

Pattern 4: Instruction Library

- Curate proven templates (`auth-api.md`, `crud-service.md`, `react-form.md`, `data-pipeline.md`) and modular `SKILL.md` bundles so new engineers ship compliant services from day one.

07. Tools, Templates & Ecosystem

Tools That Support IDD Workflows

- **Google Antigravity (`AGENTS.md`, `GEMINI.md`, `.agents/rules/`, `/plan`, `/grill-me`):** Agent-first IDE & CLI with executable Plan Artifacts, interactive design interviews (`/grill-me`), scoped rules, and native `SKILL.md` discovery
- **Claude Code (`CLAUDE.md`, `.claude/rules/`, Plan Mode):** Hierarchical instruction files (`@path` imports, path-scoped rules), deterministic lifecycle hooks, and `Shift+Tab` read-only Plan Mode
- **`google-agents-cli` (`agents-cli`):** Unified Google Cloud CLI (`uvx google-agents-cli`) for installing official agent skills, scaffolding ADK prototypes, running evaluation gates, deploying to Cloud Run / GKE / Agent Runtime, and publishing to Gemini Enterprise
- **Cursor Rules (`.cursorrules` / `.cursor/rules/*.mdc`):** Project-level instruction files that persist context across IDE sessions
- **Claude Projects (System Prompts):** Persistent workspace-level instructions and reference documents
- **GitHub Copilot Instructions:** Repository-level `.github/copilot-instructions.md` files constraining Copilot code generation
- **Aider Conventions:** `.aider.conf.yml` and convention files encoding repository standards
- **Windsurf Memories:** Persistent cross-session architecture context

Spec Storage & Quality Checklist

- Store specs in `/specs` or `/docs/idd` using `IDD-{module}-{feature}-v{version}.md` plus `SPEC_INDEX.md`, versioned in the same branch and PR as the code.
- Ensure every requirement is **testable**, free of vague words (*"appropriate," "nice," "fast"*), locked to **specific versions/libraries**, **explicit in file structure**, and **complete on error/edge cases**.
- **Avoid Over-Specification:** Keep spec depth proportional to complexity — a 50-line helper needs a few bullet points, whereas multi-service or regulated features warrant full six-part specs.

08. Agent Skills (`SKILL.md`), 3-Level Progressive Disclosure & `GCPSkillRegistry`

Dumping 40 procedural runbooks or dozens of verbose MCP tool schemas into `AGENTS.md` / `CLAUDE.md` causes **context bloat** (also called **tool bloat**): reasoning accuracy drops, latency rises, and prompt-cache costs balloon on unrelated tasks. The open **Agent Skills standard (`agentskills.io`)** solves this through **3-Level Progressive Disclosure**.

Directory Layout & Discovery Scopes

```
.agents/skills/deploy-cloud-run/  # Workspace skill (or ~/.config/agents/skills/, .claude/skills/)
├── SKILL.md  # Required: YAML frontmatter + core workflow (< 5,000 tokens)
├── scripts/  # Optional: non-interactive CLI helpers (--help, zero TTY prompts)
│   └── verify_service.py
├── references/  # Optional: deep API/IAM tables read only when needed
│   └── vpc-sc-troubleshooting.md
└── assets/  # Optional: starter templates & JSON schemas
    └── service.yaml.tmpl
```

3-Level Progressive Disclosure Architecture

Disclosure Level	What Enters Context Window	Token Cost	Trigger Condition
Level 1 — Discovery (Catalog)	YAML frontmatter only (`name` optional, **`description` required**)	**~100 tokens / skill** at session startup	Loaded automatically when the session starts
Level 2 — Instructions (Activation)	Full `SKILL.md` Markdown body	**< 5,000 tokens** recommended	Loaded *only* when user intent semantically matches `description` (or explicit `/skill-name`)
Level 3 — Execution & Deep Reference	`scripts/`, `references/`, `examples/`, `resources/`	**0 tokens until read/invoked**	Read via file tools or executed as black-box CLI commands strictly on demand

Critical Authoring Rules for `SKILL.md`

1. **`description` Is the Sole Semantic Trigger:** At Level 1, the agent *only* sees YAML frontmatter (`name` + `description`). A vague description (*"Helps with cloud stuff"*) never triggers reliably; write keyword-rich trigger conditions (*"Deploy and troubleshoot ADK services on Cloud Run with VPC-SC; trigger when editing service.yaml or debugging 403 ingress errors"*).
2. **Keep `SKILL.md` Under 5,000 Tokens & Scripts Non-Interactive:** Move bulky API tables into `references/` and deterministic validation into `scripts/` exposing `--help` flags and structured JSON/exit codes. **Never** use interactive `input()` / TTY prompts inside skill scripts — autonomous agents run headlessly and will hang.

3. **Skill Design Categories:** - **Domain Reference:** Standards, schema contracts, and API specifications referenced without file mutations - **CLI & MCP Automation:** Non-interactive command wrappers and parameter presets for linters, build tools, and MCP servers - **Template Scaffolding:** Generates boilerplate components and test harnesses from project templates - **Quality & Policy Auditing:** Evaluates code changes, configurations, or schemas against architectural and security rubrics - **Multi-Phase Execution:** Coordinates multi-stage procedures (such as dependency upgrades, migration verification, or eval runs)

4. **Enterprise Governance via `GCPSkillRegistry`:** Rather than copying `.agents/skills/` across hundreds of repos manually, platform teams publish versioned skills centrally on Google Cloud using `GCPSkillRegistry` (`Skill`, `SkillRevision`, and `Publisher` resources), enforcing IAM access control, provenance, and pinned revision rollouts across Antigravity, Gemini CLI, and ADK agents.

09. Enterprise Customizations & `agents-cli` Spec-to-Eval Workflows

Production IDD separates **always-on workspace invariants (Rules)**, **on-demand procedural playbooks (Skills)**, **zero-exception deterministic guardrails (Hooks)**, and **automated CI evaluation gates (`agents-cli`)**.

Rules vs. Skills vs. Hooks Across Antigravity & Claude Code

Mechanism	Google Antigravity	Claude Code	Enforcement & When to Use
Always-On / Path-Scoped Rules	`AGENTS.md` / `GEMINI.md` + `.agents/rules/*.md` with **strict snake_case `trigger:`** (`always_on`, `model_decision`, `glob`, `manual`; max ~24 KB/file, ~20k-token total budget)	`CLAUDE.md`, `.claude/CLAUDE.md`, `~/.claude/CLAUDE.md` (`@path` imports) + `.claude/rules/*.md` with YAML `paths:` globs	**Advisory LLM context** for repo architecture, naming, and style conventions. *(Gotcha: camelCase `alwaysOn` in Antigravity YAML is silently discarded!)*
Modular Skills (`SKILL.md`)	`.agents/skills/<name>/SKILL.md` + `~/.config/agents/skills/` + `GCPSkillRegistry`	`.claude/skills/<name>/SKILL.md` + `~/.claude/skills/`	**On-demand procedural workflows** (<5k tokens) loaded via 3-level progressive disclosure to avoid context bloat.
Deterministic Lifecycle Hooks	`hooks.json` lifecycle scripts	`settings.json` hooks (`PreToolUse`, `PostToolUse`, `Stop`, `UserPromptSubmit`)	**100% deterministic OS enforcement:** `PreToolUse` exit code `2` blocks forbidden commands/secret writes; `PostToolUse` runs `ruff`/`tsc`; `Stop` verifies tests pass.
Interactive Spec Alignment	`/plan` (executable Plan Artifact) & `/grill-me` (structured design interview before coding)	Plan Mode (`Shift+Tab` read-only exploration & plan approval)	**Pre-code human alignment** during IDD Phase 1–2 (`Define → Review`) to resolve ambiguities before touching files.

End-to-End Spec-to-Eval Automation with `agents-cli` (PAA Sec 2.1 & 4.1)

Google's `google-agents-cli` (`agents-cli`) bridges IDD specifications directly into scaffolded ADK code, automated evaluation gates, and managed cloud deployment:

```
# 1. Install official Google Cloud agent skills into Antigravity, Claude Code & Gemini CLI
uvx google-agents-cli setup

# 2. Scaffold an ADK agent from the approved IDD spec (headless: pass --prototype --yes)
agents-cli create refund-agent --prototype --yes
agents-cli install

# 3. Gate PRs in CI against golden scenarios & rubrics (evalsets configured in eval_config.json)
agents-cli eval run --config tests/eval/eval_config.json

# 4. Deploy to managed runtime (Agent Runtime, Cloud Run, or GKE) & publish to Gemini Enterprise
agents-cli deploy --deployment-target agent_runtime --no-confirm-project
agents-cli publish gemini-enterprise --agent-runtime-id <RUNTIME_ID>
```

EXAM GOTCHA — HEADLESS CODING AGENTS & CI RUNNERS (`--yes`)

Humans running `agents-cli` interactively in a terminal can answer wizard prompts, but **coding agents (Antigravity, Claude Code) and CI/CD pipelines MUST invoke `agents-cli` with `--yes` and explicit CLI flags** so non-TTY subprocesses never hang waiting on `stdin`.

10. Key Takeaways

Key Takeaways

- **IDD/SDD** writes structured instruction contracts *before* code generation (`Define → Review → Execute → Validate → Refine → Archive`).

- Every complete IDD spec covers **Context, Functional Requirements, Non-Functional Requirements, Technical Constraints, File Structure, and Acceptance Criteria**.
- **Agent Skills (`SKILL.md`)** eliminate context/tool bloat via **3-Level Progressive Disclosure**: Level 1 YAML `description` (~100 tokens) at startup, Level 2 `SKILL.md` (<5,000 tokens) on match, and Level 3 non-interactive `scripts/` & `references/` on demand — governed org-wide via **`GCPSkillRegistry`**.

- Separate **Rules** (advisory conventions; Antigravity `.agents/rules/*.md` requires snake_case `trigger: always_on | glob | model_decision | manual`), **Skills** (on-demand workflows), and **Hooks** (`PreToolUse` exit `2` / `PostToolUse` formatters for zero-exception deterministic enforcement).
- Use **`/plan`** and **`/grill-me`** (or Claude Code Plan Mode) to lock design decisions before code generation, then automate scaffolding, trajectory/rubric CI gating (`agents-cli eval run`), deployment, and Gemini Enterprise publishing with **`agents-cli`** **(`--yes` in headless mode)**.